Imperativ Programmering, tentamen

MARTIN MAGNUSSON
15 januari 2019

- Inga hjälpmedel (böcker eller annat material) är tillåtna.
- Antaganden utöver de som står i uppgiften måste anges.
- Skriv gärna förklaringar om hur du har tänkt. Även ett svar som är fel kan ge poäng, om det finns en förklaring som visar att huvudtankarna var rätt.
- Jourhavande lärare: Martin Magnusson. Martin kommer förbi cirka kl 15.30 för att svara på eventuella frågor.
- Tentan har tre delar, som svarar mot betygskriterierna för betyg 3, 4 och 5.
  - För att bli godkänd, med betyg 3, behövs 18 av 28 poäng från ”3”-frågorna (fråga 1–6) **eller** 28 poäng totalt (från alla frågor).
  - För betyg 4 behövs dessutom 9 av 14 poäng från ”4”-frågorna (fråga 7–11) **eller** 20 poäng sammanlagt från 4- och 5-frågorna (7–13).
  - För betyg 5 behövs dessutom 8 av 14 poäng från ”5”-frågorna (fråga 12–13).

Betyg 3

Fråga 1

6 poäng

Ett imperativt program består i princip av ett antal *satser* som utförs efter varandra. Satser brukar också innehålla ett eller flera *uttryck*. Varje uttryck har ett *värde*.

Vad har uttrycken i följande satser för värde?

a) `10 + 5;`

b) `10 == 5;`

Förutom att skriva program med en lång lista av satser, så kan man också använda särskilda satser för *flödeskontroll*, det vill säga loopar (för att repetera satser) och villkorlig körning med `if`-satser.

Hur många gånger skrivs bokstaven a ut i följande fall (förutsatt att kodsnuttarna ligger i ett giltigt C-program)?

c)
```
for (int i = 1; i < 5; i++)
  printf("a");
```

d)
```
for (int i = 0; i < 5; i++)
{
  if (i % 2 == 0)
    printf("a");
}
```

e)
```
int i = 5;
while (i >= 0)
{
  printf("a");
  i = i - 1;
}
```

f)
```
int i = 0;
do
{
  printf("a");
} while (i < 5);
```

Lösning

a) 15

b) 0 (eftersom $10 \neq 5$)

c) Fyra (loopar för `i` = 1, 2, 3, 4). Måsvingar behövs inte när det bara är en sats i loopen.

d) Tre (skriver ut när `i` = 0, 2, 4)

e) Sex (loopar för `i` = 5, 4, 3, 2, 1, 0)

f) Oändligt (eftersom `i` aldrig ändras)

Fråga 2

6 poäng

a) Vilken av typerna **`signed char`** och **`unsigned char`** kan innehålla störst tal?

b) *Varför* är det skillnad på hur stora tal de kan innehålla?

c) Vad har `e` och `f` för värde efter att följande kod har körts (förutsatt att kodsnutten ligger i ett komplett program)?

```
int a = 2.0;
int b = 10.0;
double c = 2.0;
double d = 10.0;
float e = a/b;
float f = c/d;
```

d) Vad skriver följande kod ut för värde på `x` respektive `a` (förutsatt att kodsnutten ligger i ett komplett program)?

```
int x = 1;
int a = 1;
for (int i = 0; i <= 10; ++i)
{
  int x = i;
  a = i;
}
printf( "%d %d", x, a );
```

e) Vilken av följande tre kodrader är giltig C-kod?

1) `char x = x;`
2) `char x = 'x';`
3) `char x = "x";`

f) Vilken datatyp är bäst lämpad för en variabel som ska lagra månadssiffran i ett datum, alltså nr 1–12?

1) **`unsigned int`**
2) **`float`**
3) **`char`**`[12]`

Lösning

a) **`unsigned char`**

b) Eftersom en bit används till att representera (minus-)tecknet i **`signed char`**. Därför är det största talet som kan representeras hälften så stort som för **`unsigned char`**.

Jag har gett 0.5 p avdrag om det inte framgår

c) 0 respektive 0.2. Eftersom `a/b` är heltalsdivision trunkeras resultatet till 0 innan det skrivs till `e`. Variabeln `f` blir däremot $2/10 = 0.2$. Även om det står flyttal på högersidan i de första två raderna är det heltal som lagras i `a` och `b`, eftersom de är av typen **`int`**.

d) `x` är 1 och `a` är 10. I loopen ändras inte den `x` som `printf` använder. I stället skapas en ny **`int`** `x` i varje varv i loopen.

e) Rad 2. Rad 1 använder en odeklarerad variable. Rad 3 försöker lagra en sträng (**`char`**`[2]`) i en **`char`**.

f) **`unsigned int`** (alternativ 1) är bäst av de här, eftersom det rör sig om heltal. Det var flera som svarade **`char`**`[12]` i stället, men det passar ju bättre för strängen "januari", till exempel.

Fråga 3

4 poäng

Skriv en algoritm som hittar det talet med det *minsta absolutvärdet* i en sekvens av tal, *utan* att använda dig av en färdig funktion som beräknar absolutvärdet. Till exempel: givet sekvensen $[-10, 20, -2, 50, 50]$ ska din algoritm returnera -2.

Du kan antingen skriva din algoritm med C-syntax eller i klartext. Om du väljer att skriva i text är det viktigt att funktionsbeskrivningen ändå är lika detaljerad som den skulle behöva vara i ett C-program. Däremot behöver inte syntaxen vara enligt C. För den här uppgiften ges inga avdrag för syntaxfel, så länge det är klart vad som är meningen ska hända. Det är inte en övning i C-programmering, men däremot i (imperativt) algoritmiskt tänkande.

Observera att du inte får använda dig av en funktion som beräknar absolutvärdet. Du ska alltså inte använda dig av notationen $|x|$ eller funktionen `abs(x)` eller `fabs(x)`.

Du kan anta att input består av en sekvens med minst ett tal.

Lösning

En lösning i klartext/pseudokod:

- Input: en lista L med tal. L innehåller minst ett element.

1) Låt $T \leftarrow$ det första elementet i L. *(T representerar det "bästa" värdet hittills.)*
2) Låt $i \leftarrow 1$.
3) Låt $n \leftarrow$ antal element i L.
4) Iterera så länge $i \leq n$:
    a) Låt $x \leftarrow$ element nummer i i L.
    b) Om $x^2 < T^2$, låt $T \leftarrow x$. *(Samma resultat som om $|x| < |T|$)*
    c) Låt $i \leftarrow i + 1$.
5) Returnera T.

Nedan är en lösning i C, som också konceptuellt är lite annorlunda. I stället för att jämföra kvadraten av talen (för att se till att båda har samma tecken) skapas här två temporära variabler i stället, som innehåller absolutvärdena av talen som ska jämföras. Algoritmen ovan skulle naturligtvis också gå bra att implementera i C.

```
double minabs( double array[], int n )
{
   double T = (array[0]);
  for (int i = 1; i < n; ++i)
  {
    // I stället för abs
    double aT;
    if (T < 0)
      aT = -T;
    else
      aT = T;
    double ax;
    if (array[i] < 0)
      ax = -array[i];
    else
      ax = array[i];

    if (ax < aT) // Hittat ett nytt minsta
    {
      T = (array[i]);
    }
  }
  return T;
}
```

Flera hade lösningar som inte returnerade negativa tal (så som det står i instruktionerna), utan i stället absolutvärdet av talen. Det har jag gett poängavdrag för.

Fråga 4

I den här uppgiften ska du skriva två C-funktioner.

4 poäng

a) Skriv en C-funktion `median3` som tar tre flyttal av typen **`double`** som argument, och som returnerar *medianen* av de tre talen.

Medianen är det mittersta värdet (inte det största och inte det minsta). Alltså: medianen av $(0,1,2)$ är 1, och medianen av $(1,2,1)$ är också 1.

2 poäng

b) Skriv också en `main`-funktion som använder din `median3`, för att testa funktionaliteten. I `main` ska du anropa `median3` tre gånger, med lämpliga värden, och skriva ut resultatet. Se till att du täcker in lämpliga värden, och att din funktion också skriver ut om testerna har lyckats eller ej.

Lösning

a) Nedan är ett kompakt sätt att lösa uppgiften på. Flera hade lösningar med en lång rad **`if`**-satser i stället för att kombinera villkoren med `&&` (and) och `||` (or) som här. Det går också bra, även om det blir ett längre program.

```
double median3(double x1, double x2, double x3)
{
  if ((x1 <= x2 && x1 >= x3)||(x1 >= x2 && x1 <= x3))
    return x1;
  else if ((x2 <= x3 && x2 >= x1)||(x2 >= x3 && x2 <= x1))
    return x2;
  else //if ((x3 <= x1 && x3 >= x2)||(x3 >= x1 && x3 <= x2))
    return x3;
}
```

b) Till exempel

```
int main()
{
  if (median3(0,1,2) == 1)
    printf("0,1,2 OK\n");
  if (median3(10,10,10) == 10)
    printf("10,10,10 OK\n");
  if (median3(1,2,1) == 1)
    printf("1,2,1 OK\n");
}
```

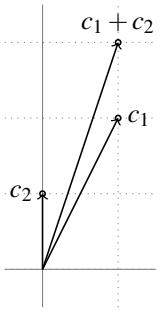
Observera att både testfall och förväntat utfall måste vara med för att få full poäng (som i exemplet ovan)!

Fråga 5

4 poäng

Markera alla rader som det är fel på i följande program. Det finns både syntaxfel (som inte går igenom kompilatorn), och logiska fel (som ger fel resultat). Det är fyra fel totalt. (För varje rad kan du anta att hela resten av programmet är korrekt. Du ska alltså inte ta hänsyn till eventuella följdfel, som följer av fel längre upp i koden.)

Skriv inte ditt svar direkt på tentan, utan på ett lösningsblad, precis som med de andra frågorna.




```
1  #include <stdio.h>
2
3  typedef {double x; double y;} coordinate2d;
4
5  coordinate2d add_2d_coordinates( coordinate2d x, coordinate2d y )
6  {
7    x->x += y->x;
8    x.y += y.y;
9    return x;
10 }
11
12 int main()
13 {
14   coordinate2d c1;
15   x = 1; y = 2;
16
17   coordinate2d c2;
18   c2.x = 0; c2.y = 1;
19
20   coordinate2d c3 = add_2d_coordinates( c1, c2 );
21
22   // En av följande rader skriver ut fel resultat:
23   printf( "c1 + c2 = [%f, %f]\n", c3.x, c3.y );
24   printf( "c1 + c2 = [%f, %f]\n", c1.x, c1.y );
25
26   return 0;
27 }
```

Lösning

a) rad 3: syntaxfel: ska vara **`typedef struct ...`**

b) rad 7: fel typ: eftersom `x` inte är en pekare kan man inte använda notationen `x->x` utan i stället `x.x`

c) rad 15: odefinierade variabler: `x` och `y` finns inte här

d) rad 24: logiskt fel: skriver ut `c1`, och inte `c1+c2` (`c1` är oförändrad, eftersom det är en kopia av den som ändras inuti `add_2d_coordinates`)

Fråga 6

2 poäng

Skriv ett C-program som läser in ett förnamn, efternamn och födelseår – ett i taget – och skriver ut dem på en rad. Till exempel:

```
Skriv in förnamn: Ada
Skriv in efternamn: Lovelace
Skriv in födelseår: 1815
Ada Lovelace 1815
```

Använd standardfunktioner från biblioteket `stdio.h`. Se till att din lösning är ett komplett program.

Lösning

```
#include <stdio.h>
int main()
{
  char firstname[20], lastname[20];
  int year;
  printf("Input your firstname: ");
  scanf("%s", firstname);
  printf("Input your lastname: ");
  scanf("%s", lastname);
  printf("Input your year of birth: ");
  scanf("%d", &year);
  printf("%s %s %d\n", firstname, lastname, year);
  return 0;
}
```

Många försökte använda **`char`** i stället för ett **`char`**-fält för att lagra namnen. Det går ju inte, eftersom **`char`** bara lagrar ett tecken, och inte en sträng! Det har jag gett 1 p avdrag för.

Betyg 4

Fråga 7

2 poäng

Demonstrera användning av operatorerna & och * genom att skriva ett C-program som skapar en variabel `int m = 30;` och skriver ut följande.

```
värdet av m : 30
adressen till m : 0x7ffd8dab4024
värdet på adressen 0x7ffd8dab4024 : 30
```

Tänk på att du måste använda både & och * i din lösning. (Pekaradresser skrivs ut med koden `%p`.)

Lösning

```
int m  = 30;
printf("värdet av m : %d\n", m );
printf("adressen till m : %p\n", &m );
printf("värdet på adressen %p : %d\n", &m, *&m );
```

Flera hade kod i stil med `int * pm = m;`, men det är ju ingen bra idé att sätta en pekaradress till *värdet* av en `int`!

Fråga 8

Något som man kan få problem med när man använder de inbyggda datatyperna i C, och liknande språk, är att ett tal plötsligt blir negativt när man räknar upp det mot större och större värden. Den här uppgiften går ut på att implementera en modul i C som är lite ”smartare” när det gäller att hantera tal, så att man slipper råka ut för fel som är svåra att hitta om ett tal har blivit ”för stort”. Anta att modulen ligger i filerna `smartint.h` samt `smartint.c`.

1 poäng

a) Deklarera en `int`-konstant som bestämmer vad som är det största talet du kan representera. (Du bestämmer själv vilket maxvärde detta ska vara.)

2 poäng

b) Skriv en funktion som adderar två heltal, och som sätter `errno=75` om resultatet blir för stort. I det fallet utförs heller ingen addition. (Det finns en variabel `errno` definierad i `errno.h`, och att sätta den till 75 representerar ”overflow”.) Tänk på att det inte fungerar att direkt kolla om summan är större än maxvärdet, eftersom summan kanske inte är korrekt om den är för stor!

2 poäng

c) Vad skulle krävas för att en användare skulle *tvingas* använda din nya säkrare adderingsfunktion, i stället för vanlig addition med operatorn ”+”? (Till exempel om man har en säkerhetskritisk tillämpning, där overflow-fel är särskilt viktiga att undvika.) Beskriv i text hur du skulle kunna justera din implementation ovan för det. Tips: det kommer inte att fungera att använda typen `int` längre.

Lösning

a) Till exempel `const int MAX = 1000;` i smartint.c eller smartint.h. Tänk på att det måste vara en *konstant*, och inte en variabel.

b)

```
1  int add( int n1, int n2 )
2  {
3    // Overflow-säkert test (förutsatt att n1
4    // och n2 var för sig är inom gränserna).
5    if (MAX-n1 < n2)
6    {
7      errno=75;
8      return n1;
9    }
10   else
11     return n1+n2;
12 }
```

Ett poängs avdrag om ett test motsvarande det på rad 5 inte finns med.

c) Då krävs att du deklarerar en egen typ och inte använder `int` i de säkerhetskritiska funktionerna. Dessutom måste du skicka runt den som en pekare till struct. Funktionen add kunde då ha följande funktionshuvud

```
add( const smartint * a,
     const smartint * b,
     smartint * summa );
```

Fråga 9

2 poäng

Om man har en variabel `char x` i C är det inte säkert om den är `signed char` eller `unsigned char`. Det beror på kompilatorn som programmet kompileras med.

Skriv ett program i C som kontrollerar om den faktiska implementationen använder `signed char` eller `unsigned char` för variabler som bara deklarerats med `char`.

Lösning

Lösningen består i att försöka lagra ett värde som ligger utanför vad som får plats i en av typerna (antingen ett negativt värde, eller ett värde större än 127), och se vad resultatet blev.

```
#include <stdio.h>
int main()
{
  char x = -1;
  if (x < 0)
    printf("chars are signed\n");
  else
    printf("chars are unsigned\n");
}
```

Fråga 10

2 poäng

Abstrakta datatyper Varför bör en abstrakt datatyp tillhandahålla en konstruktor och destruktor, enligt följande exempel?

```
typedef struct s adt;
adt* create_adt( );       // konstruktor
void delete_adt( adt* );  // destruktor
```

Lösning

Eftersom datatypen är abstrakt (inte fullständigt definierad, för vi har inte sagt vad `struct s` innehåller) går det inte att skapa en variabel av den typen här, utan bara en pekare till den. Därför måste modulen (kompileringsenheten) där typen är definierad tillhandahålla en konstruktor-funktion som skapar strukturen och returnerar en pekare till den.

Eftersom konstruktorn ofta allokerar minne på heapen (etc) behövs också en funktion som ”städar upp” och avallokerar minne som har allokerats av konstruktorn.

Dessutom kan det behövas annat initaliseringsarbete i konstruktorn, till exempel att nollställa en räknare.

För full poäng ska man ta upp två av de tre aspekterna ovan.

Fråga 11

3 poäng

Nedanstående implementation av en länkad lista (`list`) håller också koll på hur lång listan är, så att det går att skriva en funktion **`unsigned int`** `list_length(list * l)` som returnerar rätt svar omedelbart, utan att behöva traversera hela listan först. Av olika skäl kan man tänka sig att en lista har ändrats men inte uppdaterats korrekt, så att det som står i `length` inte stämmer.

Givet den här implementationen, skriv en funktion **`void`** `reindex(list * l)` som kontrollerar längden på listan och skriver rätt värde till `l->length`.

```
typedef struct n
{
  int data;
  struct n * tail;
} node;

typedef struct l
{
  node * head;
  unsigned int length;
} list;
```

Lösning

Traversera listan och beräkna längden (plus ett för varje element), och spara resultatet i `list->length`. Glöm inte att nollställa `length` i början, så att längden ser ut att öka efter varje anrop till `reindex`.

```
void reindex(list * l)
{
  l->length = 0;
  node * next = list->head
  while (next != NULL)
  {
    l->length += 1;
    next = next->tail;
  }
  return;
}
```

Betyg 5

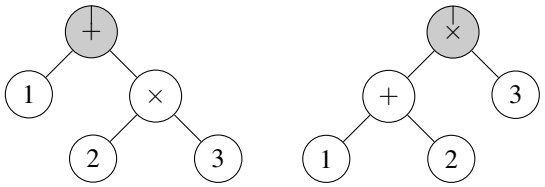
Fråga 12

Uttryck med de fyra räknesätten kan skrivas på olika sätt. Oftast skrivs de med *infix* notation, till exempel $1+2\times3$, där en binär operator som $+$ och $\times$ står *mellan* operanderna. Det går också att använda *postfix* notation, där operatorn skrivs *efter* de båda operanderna, det vill säga $1\,2\,3\,\times+$. Resultatet av uttrycket är 7 i båda fallen.

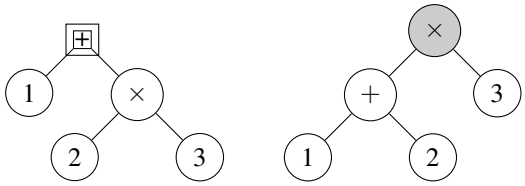
En miniräknare med postfix notation är lättare att implementera, eftersom den går att implementera med en *stack*, och eftersom det inte krävs några parenteser för tvetydiga uttryck. (Till exempel kan $1+2\times3$ tolkas som antingen $1+(2\times3)$ eller $(1+2)\times3$, medan motsvarande uttryck med postfix notation är otvetydigt: antingen $1\,2\,3\,\times+$ eller $1\,2+3\times$.)

Däremot ser det mer bekant ut för en vanlig användare om miniräknaren använder infix notation, även om det blir svårare för programmeraren som ska implementera miniräknaren. I det här fallet är det att föredra att implementera uttryck med ett *binärt träd* enligt exemplen nedan, i stället för en stack.

$1+(2\times3)$ $(1+2)\times3$



$1+(2\times3)$ $(1+2)\times3$



För att beräkna värdet av ett sådant här uttrycksträd så börjar man med den översta noden, som kallas *roten* och är skuggad i exemplen ovan. Sedan beräknar man värdena av de båda delträden som ligger under (till vänster respektive till höger) och sammanfogar dem med operatorn som ligger i rotnoden. Om en nod innehåller ett värde, och inte en operator, så beräknas värdet av noden trivialt bara genom att returnera nodens lagrade värde. Om en nod i stället innehåller en operator behöver den däremot två undernoder (en till vänster och en till höger). Värdet beräknas då *rekursivt* som (värdet av det vänstra delträdet) (operator) (värdet av det högra delträdet).

2 poäng

a) Implementera en datatyp för att representera operatorerna plus, minus, division, multiplikation. Alltså:

```
typedef /*fyll i*/ operator;
```

(Tips: Den här datatypen behöver bara användas för att välja ett av fyra räknesätt. Det ska alltså inte ske några uträkningar här.)

2 poäng

b) Implementera en datatyp för en nod i trädet. Alltså:

```
typedef /*fyll i*/ node;
```

(Vad behöver varje nod kunna innehålla för att kunna skapa träd som dem som visas ovan, när det gäller värden, operatorer, och länkar till andra noder?)

2 poäng

c) Implementera en funktion som går igenom ett träd och beräknar uttryckets värde.

2 poäng

d) Anta att varje nod i ett träd har allokerats på heapen. Implementera en funktion som, givet en pekare till roten i ett träd, frigör allt minne som allokerats av samtliga noder.

2 poäng

e) Man kan implementera den här datastrukturen antingen med bara en sorts **`struct`** för noder i trädet (på ett liknande sätt som vi gjort med länkade listor tidigare i kursen) eller med en **`struct`** som hanterar trädet, och som i sin tur innehåller pekare till en eller flera noder. Diskutera vad de här olika lösningarna har för för- och nackdelar.

Lösning

a) Här är en lösning med **`enum`** (liknande exemplet med **`enum`** för färger i en kortlek, från en av föreläsningarna), men man kan tänka sig andra lösningar också.

```
typedef enum
{
  Plus,
  Minus,
  Div,
  Mult,
  NoOp
} operator;
```

b) En nod måste ha pekare till de båda underliggande noderna, samt ett värde och en operator.

```
typedef struct n
{
  double value;
  operator op;
  struct n * left;
  struct n * right;
} node;
```

c) En rekursiv funktion som går igenom ett träd och beräknar uttryckets värde. Precis som alla rekursiva funktioner har den ett basfall, där den inte ska rekursera mer (det vill säga om vi är i en löv-nod utan några underliggande noder), och ett rekursivt fall (för andra noder).

```
1  float eval_expression( expression * f )
2  {
3    // Löv:
4    if (f->left == NULL && f->right == NULL)
5    {
6      if (f->op != NoOp)
7      {
8        // Måste ha ett värde, inte en operator, vid löv.
9        // Den här felkontrollen behöver inte vara med för full
             poäng.
10       fprintf(stderr, "Malformed expression\n" );
11       return NAN; // not a number
12     }
```

```
13      else
14        return f->value;
15    }
16    // Intern nod:
17    else
18    {
19      double left = eval_expression( f->left );
20      double right = eval_expression( f->right );
21      switch (f->op)
22      {
23      case Plus:
24        return left + right;
25      case Minus:
26        return left - right;
27      case Div:
28        return left / right;
29      case Mult:
30        return left * right;
31      default:
32        // Den här felkontrollen behöver inte vara med för full
              poäng.
33        fprintf(stderr, "Malformed expression");
34        return NAN;
35      }
36    }
37  }
```

d) En funktion som frigör allt minne som allokerats av samtliga noder. Notera att `free(f)` måste göras *efter* att de båda underträden har rekurserats, för efter `free(f)` finns inte `f->left` och `f-right` kvar.

```
1   void delete_tree( expression * f )
2   {
3     // Löv:
4     if (f->left == NULL && f->right == NULL)
5     {
6       free(f);
7     }
8     // Intern nod:
9     else
10    {
11      delete_tree(f->left);
12      delete_tree(f->right);
13      free(f);
14    }
15  }
```

e) Har man en extra struct så kan den också innehålla diverse metadata för trädet. T.ex. hur stort det är, eller ett förberäknat värde, så att man inte behöver traversera hela trädet varje gång. Däremot tar det ju lite extra minne.

Fråga 13

2 poäng

a) Vilken av följande implementationer för att beräkna $1 + 1$ är mest resurseffektiv, när

det gäller användning av minne och CPU? Kommer båda att ge korrekt resultat? Motivera ditt svar utförligt. (Ett svar utan motivering ger noll poäng.)

Program 1:

```
int main()
{
  return 1 + 1;
}
```

Program 2:

```
int add( int n1, int n2 )
{
  return n1 + n2;
}

int main()
{
  return add(1, 1);
}
```

2 poäng

b) Vilken av följande implementationer för att beräkna $1 + 1$ är mest resurseffektiv, när det gäller användning av minne och CPU? Kommer båda att ge korrekt resultat? Motivera ditt svar utförligt. (Ett svar utan motivering ger noll poäng.)

Program 1:

```
int add( int n1, int n2 )
{
  return n1 + n2;
}

int main()
{
  return add( 1, 1 );
}
```

Program 2:

```
int * add( const int * n1,
    const int * n2 )
{
  int * result = malloc(
      sizeof( int ) );
  *result = *n1 + *n2;
  return result;
}

int main()
{
  int * sum;
  int n1 = 1;
  int n2 = 1;
  sum = add( &n1, &n2 );
  return *sum;
}
```

Lösning

a) Program 1 är mer resurseffektivt, både när det gäller minne och CPU. Program 1 måste allokera en *stack frame* för funktionsanropet, och kopiera värdet på parametrarna `n1` och `n2`.

b) Program 1 är mer resurseffektivt, både när det gäller minne och CPU. Det utför färre instruktioner, och slipper ödsla tid på att anropa `malloc`, som också tar tid. Att dereferera pekare kan också ta tid. Program 1 är också minnessnålare. Dels använder det färre variabler, och dels är variablerna (antagligen) mindre. Med den kompilator och den dator vi har använt på labbarna i kursen (GCC och 64-bitars Linux) så tar en pekare upp 64 bitar (= 8 bytes) men en **`int`** tar bara upp 32 bitar (= 4 bytes).

Båda ger korrekt resultat.